Estructura de Datos

Eficiencia de Algoritmos

Ingeniería en Computación e Informática

Tabla de contenidos

1. Introducción
2. Conceptos de eficiencia
3. Análisis de Algoritmos
4. Calcular el orden de complejidad

Universidad Andrés Bello

Introducción

- Un algoritmo es un conjunto de instrucciones claramente especificadas que el computador debe seguir para resolver un problema.

- Una vez que se ha dado un algoritmo para resolver un problema y se ha probado que es correcto, el siguiente paso es determinar la cantidad de recursos, tales como tiempo y espacio, que el algoritmo requerirá para su aplicación.

- Un algoritmo que necesita varios gigabytes de memoria principalmente no es útil en la mayoría de las maquinas actuales.

Características de un algoritmo

1. **Entrada :** definir lo que necesita el algoritmo.
2. **Salida :** definir lo que produce.
3. **No ambiguo :** explícito, siempre sabe qué comando ejecutar.
4. **Finito :** El algoritmo termina en un número finito de pasos.
5. **Correcto :** Hace lo que se supone que debe hacer. La solución es correcta.
6. **Efectividad**: Cada instrucción se completa en tiempo finito. Cada instrucción debe ser lo suficientemente básica como para que en principio pueda ser ejecutada por cualquier persona usando papel y lápiz.

Razones para Estudiar los Algoritmos

- Supongamos que se dispone, para resolver un problema dado, de un algoritmo que necesita un tiempo exponencial y que, en un cierto computador, una implementación del mismo emplea $10^{-4} \times 2^n$ segundos.
- Resuelve un problema de tamaño $n = 10$ en una décima de segundos.
- Entonces necesitaremos casi 10 minutos para resolver uno de tamaño 20.
- Un día no bastara para resolver uno de tamaño 30.
- En un año de cálculo ininterrumpido, alcanzaremos uno de tamaño 38.

Razones para Estudiar los Algoritmos

- Compramos un computador cien veces más rápida.
- El mismo algoritmo conseguirá resolver ahora un ejemplar de tamaño n en sólo $10^{-6} \times 2^n$ segundos.
- ¡Qué decepción al constatar que, en un año, apenas se consigue resolver un ejemplar de tamaño 45!.

Razones para Estudiar los Algoritmos

- Imaginemos, en cambio, que investigamos y encontramos un algoritmo capaz de resolver el mismo problema en un tiempo cúbico.
- La implementación de este algoritmo en el computador inicial podría necesitar, por ejemplo, $10^{-2} \times n^3$ segundos.
- En un día un ejemplar de un tamaño superior a 200.
- Un año permitiría alcanzar casi el tamaño 1500.
- Por tanto, el nuevo algoritmo no sólo permite una aceleración más espectacular que la compra de un equipo más rápido, sino que hace dicha compra más rentable.

Conceptos de eficiencia

- Un algoritmo es **eficiente** cuando logra llegar a sus objetivos planteados utilizando la menor cantidad de recursos posibles, es decir, minimizando el uso memoria, de pasos y de esfuerzo humano.
- Un algoritmo es **eficaz** cuando alcanza el objetivo primordial, el análisis de resolución del problema se lo realiza prioritariamente.
- Puede darse el caso de que exista un algoritmo eficaz pero no eficiente, en lo posible debemos de manejar estos dos conceptos conjuntamente.

Conceptos de eficiencia

- La eficiencia de un programa tiene dos ingredientes fundamentales: **espacio** y **tiempo**.
  - La *eficiencia en espacio* es una medida de la cantidad de memoria requerida por un programa.
  - La *eficiencia en tiempo* se mide en términos de la cantidad de tiempo de ejecución del programa.
- Ambas dependen del tipo de computador y compilador, por lo que no se estudiará aquí la eficiencia de los programas, sino la eficiencia de los algoritmos.
- El análisis dependerá de si trabajamos con máquinas de un solo procesador o de varios de ellos. Centraremos nuestra atención en los algoritmos para máquinas de un solo procesador que ejecutan una instrucción y luego otra.

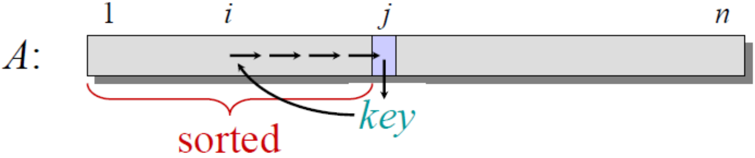
El problema de ordenamiento

1. **Input**: una secuencia $\langle a_1, a_2, ..., a_n \rangle$ de números.
2. **Output**: una permutación $\langle a'_1, a'_2, ..., a'_n \rangle$ tal que $a'_1 \leq a'_2 \leq ... \leq a'_n$.
3. Ejemplo:
   - Input: 8 2 4 9 3 6
   - Output: 2 3 4 6 8 9

Insertion Sort

“pseudocode”

```
INSERTION-SORT (A, n)        ▷ A[1 . . n]
    for j ← 2 to n
        do key ← A[j]
           i ← j – 1
           while i > 0 and A[i] > key
               do A[i+1] ← A[i]
                  i ← i – 1
           A[i+1] = key
```



Ejemplo de Insertion Sort

8	2	4	9	3	6	
2	8	4	9	3	6	
2	4	8	9	3	6	
2	4	8	9	3	6	
2	3	4	8	9	6	
2	3	4	6	8	9	*done*

Tiempo de Ejecución

1. El tiempo de ejecución depende de la entrada: una entrada ya ordenada es más fácil de ordenar.
2. Se debe parametrizar el tiempo de ejecución con el tamaño de la entrada, ya que secuencias más cortas son más fáciles de ordenar que secuencias más largas.
3. Generalmente, analizamos los cotas superiores del tiempo de ejecución, ya que a todos nos gusta una garantía.

Clases de análisis

1. Peor caso: (usualmente)
   - $T(n) =$ máximo tiempo del algoritmo bajo cualquier entrada de tamaño n.
2. Caso promedio: (algunas veces)
   - $T(n) =$ tiempo esperado del algoritmo bajo todas las entradas de tamaño n.
   - Necesita supuestos de la distribución estadística de las entradas.
3. Mejor caso: (casi nunca)
   - Truco con un algoritmo lento que corre rápido sobre *alguna* entrada.

Tiempo independiente de la máquina

1. ¿Cuál es el tiempo del peor caso de *Insertion Sort*?
   - Depende de la velocidad de nuestro computador:
   - Velocidad relativa (en la misma máquina),
   - Velocidad absoluta (sobre diferentes máquinas).

2. **GRAN IDEA** :
   - Ignorar las constantes independientes de la máquina.
   - Analizar el **crecimiento** de $T(n)$ como $n \to \infty$.

Análisis Asintótico

Tiempo independiente de la máquina

1. ¿Cuál es el tiempo del peor caso de *Insertion Sort*?
   - Depende de la velocidad de nuestro computador:
   - Velocidad relativa (en la misma máquina),
   - Velocidad absoluta (sobre diferentes máquinas).

2. **GRAN IDEA** :
   - Ignorar las constantes independientes de la máquina.
   - Analizar el **crecimiento** de $T(n)$ como $n \to \infty$.

Análisis Asintótico

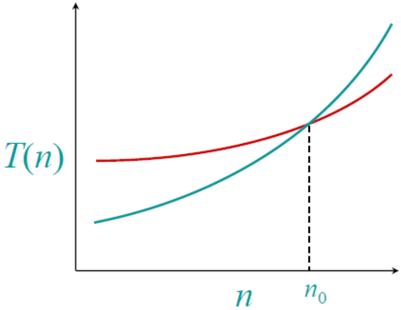
Notación Θ

1. Matemática:
   - $\Theta(g(n)) = \{f(n)$: existen las constantes positivas c_1, c_2, y n_0 tal que $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$ para todo $n \geq n_0\}$.
2. Ingeniería:
   - Eliminar los términos de bajo orden; ignorar las constantes restantes.
   - Ejemplo: $3n^3 + 90n^2 - 5n + 6046 = \Theta(n^3)$.

Desempeño Asintótico

1. Cuando n es suficientemente grande, un algoritmo $\Theta(n^2)$ siempre supera a un algoritmo $\Theta(n^3)$.

- Sin embargo, no deberíamos ignorar los algoritmos asintóticamente más lentos.
- En las situaciones de diseño del mundo real, a menudo se realiza un balanceo cuidadoso de los objetivos ingenieriles.
- El análisis asintótico es una herramienta útil para ayudar a estructurar nuestro pensamiento.



Análisis de Insertion Sort

1. Peor Caso : entrada ordenada en forma inversa.

   $$T(n) = \sum_{j=2}^{n} \Theta(j) = \Theta(n^2) \text{ [serie aritmética]}$$

2. Caso Promedio : todas las permutaciones son igualmente probables.

   $$T(n) = \sum_{j=2}^{n} \Theta(j/2) = \Theta(n^2)$$

3. ¿Es Insertion Sort un algoritmo rápido de ordenamiento?
   - Moderadamente, para n pequeños.
   - No, para n grandes.

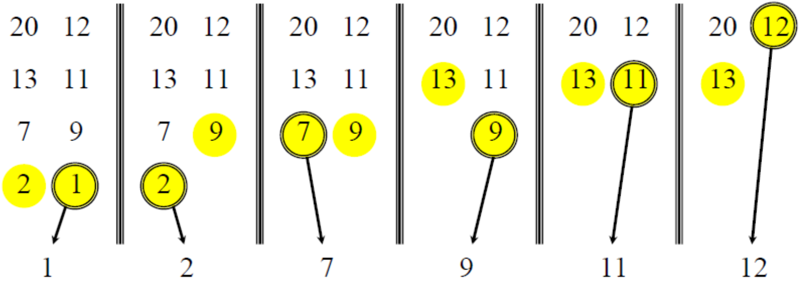
Merge Sort

MERGE-SORT $A[1\,..\,n]$

1. If $n = 1$, done.
2. Recursively sort $A[\,1\,..\lceil n/2 \rceil\,]$ and $A[\,\lceil n/2 \rceil{+}1\,..\,n\,]$.
3. "*Merge*" the 2 sorted lists.

Subrutina Clave: **MERGE**

"Merge" de dos arreglos ordenados



Tiempo $= \Theta(n)$ para hacer el merge de un total de n elementos (tiempo lineal).

Universidad Andrés Bello

Análisis de Merge Sort

$T(n)$	**MERGE-SORT** $A[1\,..\,n]$
$\Theta(1)$	1. If $n = 1$, done.
$2T(n/2)$	2. Recursively sort $A[\,1\,..\,\lceil n/2\rceil\,]$ and $A[\,\lceil n/2\rceil + 1\,..\,n\,]$.
$\Theta(n)$	3. ***"Merge"*** the 2 sorted lists

Abuso

Dejadez: debería ser T(⌈n/2⌉)+T(⌊n/2⌋), pero no importa asintóticamente.

Recurrencia de Merge Sort

$$T(n) = \begin{cases} 1 & \text{si } n = 1 \\ \\ 2T(n/2) + n & \text{si } n > 1 \end{cases}$$

1. Usualmente omitiremos la definición del caso base cuando $T(n) = \Theta(1)$ para n suficientemente pequeños, pero sólo cuando no tenga efecto en la solución asintótica de la recurrencia.

Como calcular la complejidad

- Algoritmos Iterativos.
- Algoritmos Recursivos.

Calcular Complejidad Computacional

Algoritmo Iterativo

Algorithm 1 Factorial

Require: n: número a calcular
Ensure: factorial

```
1: if n = 0 o n = 1 then
2:     Fact ← 1
3: else
4:     for j ← 2 a n do
5:         Fact ← Fact × j
6:     end for
7: end if
8: return Fact
```

El análisis se realiza línea por línea:

1 Instrucción condicional (Si) y asignación con tiempo 1.

2 Instrucción condicional (caso contrario):

3 Asignación con tiempo 1

4 Ciclo que se ejecuta $n - 1$ veces dentro del ciclo:

5 Asignación con tiempo 1.

8 Retornar con tiempo 1.

Con los tiempos indicados, se tiene:

- $C(FACT) = 1 + 1[1 + (n - 1) \times 1] + 1$

Análisis factorial

Con los tiempos indicados, se tiene:

- $C(FACT) = 1 + 1[1 + (n - 1) \times 1] + 1$

Simplificando:

- $C(FACT) = 1 + 1 + (n - 1) + 1$

Aplicando propiedades de O: $C(FACT) = O(1) + O(1) + O(n - 1) + O(1)$ y como O desprecia las constantes aditivas y multiplicativas, se tiene finalmente que:

- $C_{PC}(FACT) \sim O(n)$

Calcular Complejidad Computacional

Algoritmo Recursivo

Algorithm 2 Factorial

Require: n: número a calcular.
Ensure: factorial del número.

```
if n = 0 o n = 1 then
    return 1
else
    return n× Factorial (n − 1)
end if
```

Para calcular el factorial, se realiza una ecuación de recurrencia:

$$T(n) = \begin{cases} 1 & \text{si } n = 1 \\ T(n-1) + 1 & \text{si } n > 1 \end{cases}$$

Como resolver una recurrencia

$$T(n) = \begin{cases} 1 & \text{si } n = 1 \\ T(n-1) + 1 & \text{si } n > 1 \end{cases}$$

$$\begin{aligned}
T(n) &= T(n-1) + 1 \\
&= (T(n-2) + 1) + 1 \\
&= T(n-2) + 2 \\
&= ((T(n-3) + 1) + 1) + 1 \\
&= T(n-3) + 3 \\
&\vdots \\
&= T(n-k) + k \\
&\vdots \\
&= T(1) + n - 1 \\
&= 1 + n - 1 \\
&= n
\end{aligned}$$

Entonces la complejidad del algoritmo es $\sim O(n)$

Universidad Andrés Bello

Ahora resolvamos

La recurrencia de Merge Sort

$$T(n) = \begin{cases} 1 & \text{si } n = 1 \\ 2T(n/2) + n & \text{si } n > 1 \end{cases}$$

$$\begin{aligned}
T(n) &= 2T\left(\frac{n}{2}\right) + cn \\
&= 2\left[2T\left(\frac{n}{4}\right) + c\left(\frac{n}{2}\right)\right] + cn \\
&= 4T\left(\frac{n}{4}\right) + cn + cn \\
&= 4\left[2T\left(\frac{n}{8}\right) + c\left(\frac{n}{4}\right)\right] + cn + cn \\
&= 8T\left(\frac{n}{8}\right) + cn + cn + cn \\
&\vdots \\
&= 2^k T\left(\frac{n}{2^k}\right) + kcn \\
&\vdots \quad k/2^k = 1 \Rightarrow k = \log_2 n \\
&= 2^{\log_2 n} T\left(\frac{n}{2^{\log_2 n}}\right) + cn \log_2 n
\end{aligned}$$

Ahora resolvamos

La recurrencia de Merge Sort

$$T(n) = \begin{cases} 1 & \text{si } n = 1 \\ 2T(n/2) + n & \text{si } n > 1 \end{cases}$$

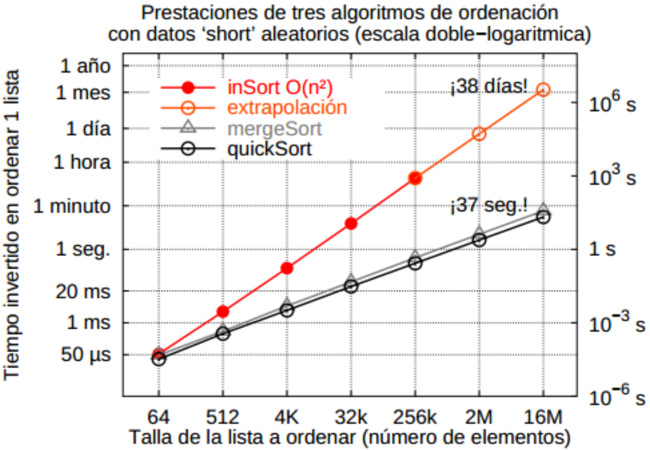
$$\begin{aligned} T(n) &= 2^{\log_2 n} T\left(\frac{n}{2^{\log_2 n}}\right) + cn\log_2 n \\ &= nT\left(\frac{n}{n}\right) + cn\log_2 n \\ &= nT(1) + cn\log_2 n \\ &= 1 + cn\log_2 n \\ &= O(n\log n) \end{aligned}$$

Conclusión

1. $O(nlogn)$ crece más lentamente que $O(n^2)$.
2. Entonces, Merge Sort es mejor asintóticamente que Insertion Sort en el peor caso.
3. En la práctica, Merge Sort es mejor que Insertion Sort para $n > 30$.
4. Como tarea, probarlo por Uds. mismos!

Conclusión

Comparación de Algoritmos de Ordenación



Universidad Andrés Bello